# NAVAL POSTGRADUATE SCHOOL
## Monterey, California




# THESIS

AN INTERACTIVE ENVIRONMENT FOR
THE DEVELOPMENT OF
AN EXPERT SYSTEM IN ZOG

by

Dempsey Butler, III
June 1984

Thesis Advisor: Bruce J. MacLennan



Approved for public release; distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER | 2. GOVT ACCESSION NO. A149954 | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle) An Interactive Environment for the Development of an Expert System in ZOG | | 5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June 1984 |
| | | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s) Dempsey Butler, III | | 8. CONTRACT OR GRANT NUMBER(s) |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943 | | 12. REPORT DATE June 1984 |
| | | 13. NUMBER OF PAGES 62 |
| 14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) | | 15. SECURITY CLASS. (of this report) UNCLASSIFIED |
| | | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Expert System, Frame, Human-Computer Interface, OPS7, Schema, ZOG

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ZOG is a rapid-response, large-network, menu-selection human-computer interface implemented on the PERQ microcomputer. This thesis develops a framework for and discusses issues relative to implementing the OPS7 expert system language as an interactive programming environment in ZOG. It begins by tracing the history of the ZOG system. The logical and physical aspects of ZOG's frame structure are explained. A discussion of the expert system language used in ZOG, OPS7, is presented to acquaint (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601



SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT (Continued)

the reader with its character. The subnet schemas required to run an OPS7 style interpreter agent are developed and the user's perspective of the agent is presented. Finally, recommendations for future work in this area are made.

Approved for public release; distribution unlimited.

# An Interactive Environment for the Development of an Expert System in ZOG

by

Dempsey Butler, III
Lieutenant, United States Navy
B.S., United States Naval Academy, 1977

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1984





Author: Dempsey Butler, III

Approved by: Bruce J. Lineman
Thesis Advisor

R. McLeod
Second Reader

David K. Hsiao
Chairman, Department of Computer Science

Kneale T. Marshall
Dean of Information and Policy Sciences

## ABSTRACT

ZOG is a rapid-response, large-network, menu-selection human-computer interface implemented on the PERQ micrccomputer. This thesis develops a framework for and discusses issues relative to implementing the OPS7 expert system language as an interactive programming environment in ZOG. It begins by tracing the history of the ZOG system. The logical and physical aspects of ZOG's frame structure are explained. A discussion of the expert system language used in ZOG, OPS7, is presented to acquaint the reader with its character. The subnet schemas required to run an OPS7 style interpreter agent are developed and the user's perspective of the agent is presented. Finally, recommendations for future work in this area are made.


Additional keywords: agent information systems, Shipboard aircraft carriers, AIRPLAN rule based expert system programming languages.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

# I. ZOG BACKGROUND

## A. INTRODUCTION

One of the first things a prospective computer user learns is that interaction with a machine is required if the user expects to realize the potential power of the computer. This interaction takes place via a human-computer interface. Depending on the design of the interface, there are varying degrees of usefulness which the user can achieve. It is safe to say that the more familiar one is with the interface the more computer power one has available. Suppose that the knowledge required to become familiar with the interface was embedded within the interface. If such an interface was also simply structured and responded rapidly to commands, it would allow the user to quickly understand the power of the computer. ZOG is such an interface.

ZOG appears as a rapid-response, large-network, menu-selection human-computer interface. [Ref. 1 : p.1] The basic data structure is a frame, which contains textual information and selection information about the related items. Tens of thousands of related frames exist in hierarchical networks called ZOGNETS. Selections allow the rapid traversal of ZOGNETS, the editing of frames, or the execution of programs.

## B. HISTORY OF THE ZOG PROJECT

ZOG has its origin in 1972, when a group of cognitive psychologists gathered at Carnegie-Mellon University to

investigate computer program simulations.[1] In particular, this group was interested in devising a method of using large scale simulations without prior knowledge of the programs or of the operating systems on the computers which ran the simulations. Three individuals (A. Newell, G. Robertson, and D.M. McCracken) developed a uniform interface for these programs, but the first ZOG was short lived because of the limitations in terminal technology; 300 baud hardcopy was hardly rapid response.

In 1975 Newell and Robertson served on a technical advisory committee for a system called PROMIS (Problem Oriented Medical Information System) implemented at the University of Vermont Medical School. PROMIS turned out to be remarkably similar to ZOG and it utilized terminal technology which provided a response on the order of .5 seconds.

This experience rekindled interest in ZOG, and in 1975 the Office of Naval Research (ONR) began support of a small effort to further develop ZOG into an interesting interface. Several versions of ZOG exist on machines like the PDP 10, VAX/11-780, and on the PERQ microcomputer. While developing ZOG on the minis, possible alternate implementations and difficulties regarding hardware and operating system constraints were explored. The desire to use the PERQ as the ultimate target machine was influenced by the parallel development of SPICE (scientific personal integrated computing environment) on the PERQ at Carnegie-Mellon. Planning included the use of some of the results of the SPICE research in the ZOG implementation.

---

[1]The historical information in this section is based on R.M. Akscyn, D.L. McCracken, The ZOG Project, Computer Science Department, Carnegie-Mellon University, 5 January 1983.

In 1980, Captain Richard Martin, USN, Commanding Officer for the commissioning crew of the USS CARL VINSON, visited the ZOG project. The Captain had previously decided to incorporate computer science research to make CARL VINSON a test bed for leading scientific technologies in the fleet. After visiting many CNR research sites, he believed that ZOG would meet his goals for creating an onboard testbed for further research. Although the ZOG group had not envisioned the application of ZOG in such a real-time, large scale environment, the advantages of placing ONR sponsored research into the fleet were too great to pass up.

The current ZOG-based management information system onboard CARL VINSON has a data base distributed over 28 PERQ computers. Applications cover four main areas: (1) an on-line Ship's Organization and Regulations Manual (SORM); (2) an interactive task management system which can use the SORM to decide how tasks are to be performed; (3) a rule based expert system tc aid Air Operations in the launch and recovery of aircraft; (4) an on-line training manuals which interact with videodisk display units; and (5) an electronic mail system.

Because this thesis is involved with enhancing the development environment for an expert system in ZOG, the focus will be on the third item.

## C. AIRPLAN: AN EXPERT SYSTEM IN ZOG

The USS CARL VINSON is an aircraft carrier and as such spends a substantial amount of time in the launch and recovery of aircraft. AIRPLAN is a rule-based expert system used as a decision tool for air operations officers in the launch and recovery evolution. The system is implemented in OPS7, a rule-based language, with ZOG used as the human-computer interface.

From the beginning, the development of AIRPLAN has been incremental; a kernel of the expected system was put into the operational environment, and the environment has and continues to direct the direction of growth. In support of this incremental strategy, the system allows queries about its data base and its behavior. "This ability to ask the system questions about its reasoning is useful for system developers trying to track down bugs, and for system users to both gain confidence in the abilities of the system and the correctness of its recommendations, and for eliciting additional or more detailed information on which to base decisions." [Ref. 2 : pp.2-3]

# II. INTRODUCTION

## A. A PROGRAMMING ENVIRONMENT

What is a programming environment? To answer this question it is necessary to understand what is involved in the process of writing a computer program. Initially some problem specification must exist. With such a specification, an algorithm which appears to satisfy it is found. Given these two items, the algorithm must be translated into source code, executed and debugged. The cycle of code writing, execution and debugging continues until the human programmer is convinced to some predetermined degree that the program satisfies the problem specification. The task of managing all associated files and programs falls upon the programmer.

Experience shows that this process is composed of many activities which are tedious, repetitive and detailed to the point where errors are commonplace. Examples of areas which create such problems include mastering a programming language syntax for creating and editing programs, and managing the compile/link/load process. It seems appropriate that the computer should be counted on to perform these kinds of mechanical tasks, which it excels at, while the programmer concentrates on cognitive ones. To this end, the programmer should have sufficient tools available on the computer to automate these activities [Ref. 4 : pp. 4-5]. In such an environment, creating and maintaining programs can be the responsibility of a syntax-directed editor. The process of compile/link/load can be reserved for final production programs, since in the interactive environment an interpreter is better suited for program development.

The above is just an example of what an interactive programming environment can provide. For the purposes of further discussion, the following definition of an interactive programming environment will be adopted:

> First, within a unified framework, they provide a large set of tools, most of which are specific to a particular programming language. Second, they take advantage of the fact that programs have an underlying structure that is more than a string of characters, using this structure as an organizational tool. Third, they support incremental program development, in both the design and maintenance activities. Finally, they are highly interactive in nature, promoting and exploiting a fairly high bandwidth of communication between the user and the environment [Ref. 3].

## B. THE ZOG ENVIRONMENT

The natural question which follows is, how does ZOG measure up to this more formal definition of an interactive programming environment? The notion of a frame in a tree structure satisfies the requirement of a unified framework. In ZOG, the frame is used as a visual representation of the data base for the user to see and manipulate; at the same time the frame is the structure used to store data in memory. These two separate ideas work this way. In memory, data is store in a complex PERQ PASCAL [2] record structure. For the user, the notion of a window frame exists. When the user requests to see data in memory, the data is unparsed into different fields of the window and presented on the screen.

---

[2]PERQ PASCAL is an extension of PASCAL which, among other features, supports high level string manipulation. The copyright of this extension is held by Three Rivers Computer Corporation, Pittsburgh, PA.

The second point of using the underlying structure of a language as an organizational tool is present if one considers the language to be Pascal like. The inherent structure of Pascal encourages hierarchical stepwise development and modular design. ZOG develops its ZOGNETS is just this way. The frame at the top of a subnet contains its central theme. Options are created on this frame, and these options link to other frames which further explain the central theme. But this use of the underlying structure of Pascal seems to end here. There is little evidence to show that the developers of ZOG planned to support any particular programming language from within the environment. The lack of any programming tools, such as interpreters, compilers, or syntax directed editors, makes this manifest.

The ability to support incremental program development is one area where ZOG falls short. Currently it supports development of Pascal programs by writing the programs as text in frames and then running an agent ( a program executed from within ZOG which manipulates frames) which strips the text off the frames and creates a text file out in the machine's operating system. What is needed is a method of executing parts of the program while still in the environment; this is the focus of this thesis.

The bandwidth of communication which ZOG currently has is highly interactive and user friendly. Users find that movement around the subnets is intuitive and easily mastered. Straight-forward system utilities exist for the creation and modification of frames in the data base; time on the system rapidly makes the user comfortable with these utilities.

# III. ZOG FRAME STRUCTURE

Thus far, ZOG has been viewed from a logical perspective. This chapter will expound on this view and address the physical implementation of the ZOG frame in Perq Pascal. The intent is not to make the reader an expert in using ZOG, manipulating ZOG frames, or generating code in this version of Pascal. Rather it is hoped that the reader will gain a respect for the power and complexity of this environment.

## A. THE LOGICAL VIEW

The logical view has been defined as the user's window into ZOG. Understanding this view requires an understanding of the basic parts of a ZOG frame and how they are put together.

The hierarchical arrangement of frames in ZOG is in the form of a tree. This structure, called a net, has a root frame (called the top frame), and branches, or connecting frames. As implemented on the Perq Microcomputer, ZOG is one large net composed many subordinant nets, called subnets. Inherent to the net are different levels of information: the top frame of the net contains general information and points to frames with more specific information. This pattern continues until the most specific frames in the net, the leaves, are reached. The point is that every frame describing a particular aspect of the more general subject has a place in the hierarchy of the tree that is dictated by the logical structure of the subject matter [Ref. 5 : p.11].

Every frame is divided into four component types called items: the frameid (frame identification), frame title, frame text, and selections (see figure 3.1). Selections, which represent choices of what to do next, are of three types: options, local pads, and global pads [Ref. 5 : p.12].

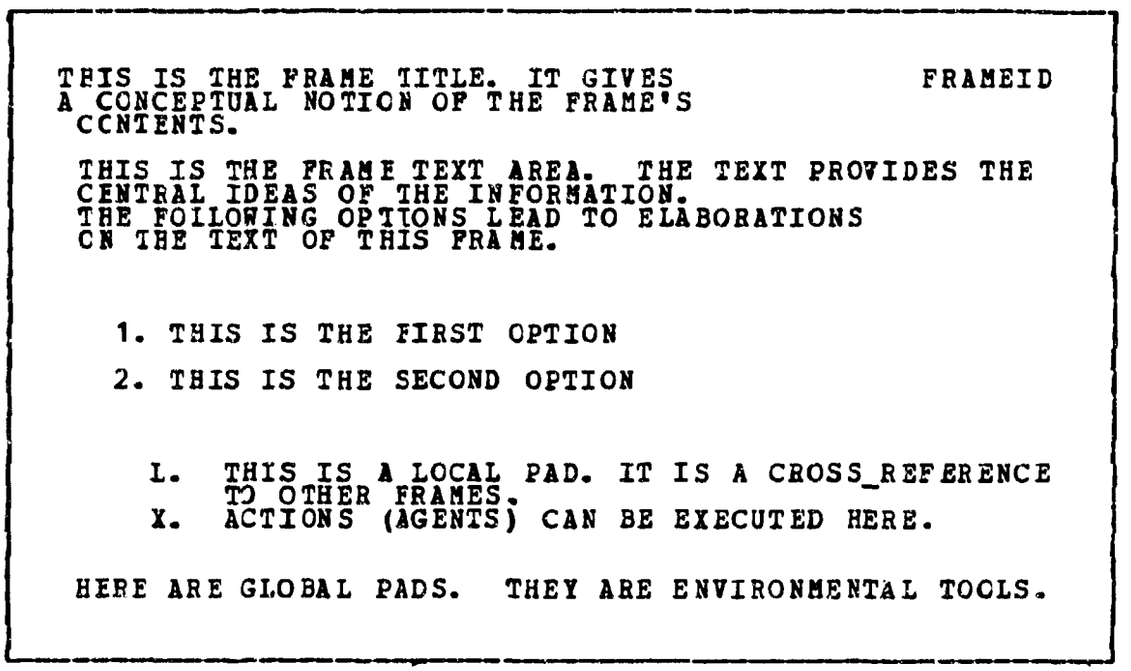


**Figure 3.1 Frame Layout.**

The frameid is the unique system label for every frame in the ZOG net. It contains the subnet name and frame number of that subnet. Frames in the same subnet have the same frameid name.

The frame title is usually the text of the option which points to it. It can be considered the conceptual link between a frame and its parent. The text of the frame can be anything the user wants.

Selections are used to point to other branches in a net. The first type of selection is an option. It consists of a selection character and text. Options are used to point to frames that are logically more specific than the frame containing the option.

The second type, the local pad, also consist of a selection character and text. While the difference between local pads and options are negligible, local pads usually cross-reference other frames rather than following a strictly logical path.

The final type of selection is the global pad. These are found across the bottom line of the frame and can be utilized by typing the first letter (always lowercase) of the desired pad. Global pads provide more choices for the user: more ways to move around nets, information about the frame's history, and utilities for tasks such as creating or deleting frames.

One other part of a frame is the user display. Zog communicates with the user on the last line of the frame, directly above the global pads. The display helps the user by suggesting what to do next, why a command from the user was not accepted, or where or not the editor is currently envoked [Ref. 5].

When frames in ZOG are created, they must originate from some frame schema. A schema is the generic frame for a subnet, and is created when a user elects to create a new subnet. The user designs the frame with anything on it, from options or text, to any number of local or global pads. This frame will now exist in ZOG as the zero frame in the subnet specified by the user. Subsequently, whenever another frame is created in this subnet, the default frame schema to be created will be the zero frame for the subnet. The option exists to use another frame schema, if desired.

## B. THE PHYSICAL VIEW

In reality ZOG is simply a very large computer program (over 70,000 lines of source code). The ZOG system is based on the record structures provided by Pascal. A frame is a record containing as many fields as there are parts to the logical frame. Some of the parts are easily recognized, such as the frame title, the text, and the options. Others are less obvious, such as the frame owner(s), the frame protection, and the action hidden behind global pads.

The field type declarations vary, depending of the nature and quantity of information that the field holds. For example, the frame ID field is simply a string of no more than 15 characters. The text field is more complex because its data may be up to 21 lines of information (double sized frames exists, and these could hold more text). To handle this, its field in the frame record points to another record, which points to a linear, doubly linked list; each element of the list contains a line of text. The source code for the frame structure can be found in appendix A.

Just how is data stored into the ZOG database? By using the utilities provided by the system, the user can create a blank frame or a new subnet of frames into which data can be stored. The ZOG editor (ZED) is used to insert information into the various fields in the frame. Once the frame is saved, ZOG parses the different fields of the display frame (called the window frame) and stores the data into the physical record frame. The retrieval of data follows the reverse path. After telling ZOG which frame you wish to see, it finds the physical record and unparses its fields into the window frame. It is interesting to note that the retrieval of a specific frame over the Ethernet takes under 1.2 seconds. If the frame is located on the same machine as the user, retrieval takes .5 seconds!

## C. SUMMARY

From the perspective of an interactive programming environment both the logical and physical views are important. The logical view of the frame provides an intuitive understanding of stored data as well as a mechanism for input and output for programs. The physical view provides the knowledge base required to design a tool in the environment.

## IV. AN EXPERT SYSTEM LANGUAGE: OPS7

Because OPS7 is the expert system currently used by ZOG, a discussion of the language is appropriate. OPS7 is a member of the OPS family of production system languages designed by Charles L. Forgy. Production systems represent a model of computation equal in power to, but very different in style from, procedural languages like Pascal, operator-oriented languages like APL, and applicative languages such as Lisp. This discussion will highlight the language's data structure, basic control structure, and conflict resolution scheme. Having this understanding will reveal the characteristics which lend OPS7 to integration within ZOG. The BNF (Backus-Naur Form) syntax for the language can be found in appendix B.[3]

### A. WORKING MEMORY ELEMENTS

The only data structure used in OPS7 is the working memory element. It is similar in form and function to Pascal's record structure. Fields in a working memory element can hold scalar values, vectors, or sets. The following is an example of a type declaration, a function call 'MAKE,' used to create an instance of the type in working memory, and a call to display a working memory element. Comments in OPS7 start with a semi-colon and terminate at the end of the line.

```
(type task
  kind = scalar     status = set:1      values = vector:3
)
```

---

[3]The bulk of the information is this chapter comes from references 5 and 6.

```
; This function creates an instance of
; type task in working memory.

(make task
  kind = sort  status = { on }     values = [ 1 2 3 ]
)

(wme 1)          ; Call to display Working Memory Element 1
                 ; What follows is the output.
                 ; (Assumes this instance of TASK is
                 ; working memory element 1.)
  task
     *id* = 1
     kind = sort
   status = { on }
   values = [ 1 2 3 ]
```

Scalar types can be either integers or symbolic atoms. Symbolic atoms may be any string of characters other than integers or anything enclosed by double quotes. Floating point operations are not implemented in OPS7. Sets are defined as unordered collections of non-repeating scalar values. Curly braces delineate sets. Vectors are ordered collections of scalar values which may repeat.

One must think of the working memory of an OPS7 program as the knowledge base about the state of a problem. It contains instances of the declared types, which are created by the MAKE function, altered by the MODIFY function, and deleted by the REMOVE function. Working memory is constantly changing. It is this change which causes the expert system to transition from one state to another.

## B. RECOGNIZE AND ACT CYCLE

The basic control structure of any production system is the recognize and act cycle. On every cycle of the OPS7 interpreter, an attempt is made to satisfy at least one left hand side of a production rule with elements from the working memory. This process defines a set of unique instances in which left hand sides of productions may be satisfied on each cycle. From here the conflict resolution

scheme must determine which instance from this conflict set is suitable for firing. The following pseudo code for this cycle is found in reference 5:

```
loop
 RECOGNIZE:
 determine the current set of instantiations;
 if the set of instantiations is empty, then halt;
 ACT:
 select 1 instantiation and execute its right hand side
actions

repeat
```

## C. CONFLICT RESOLUTION

The purpose of the conflict resolution strategy is to select the next rule to fire. Ideally the execution of rules would be order independent so that such a strategy would not be required. But in reality such a set of rules rarely exists. Due to the nature of expert systems, the conditions for different rules will be similar. And as the working memory elements are created, modified, and deleted rules have a tendency to fire sequentially although that may not have been the original plan.

Some strategy must exist to determine which instantiation is to be selected from the conflict set if the set contains more than one element. In OPS7, conflict resolution is either special case first or most recent first. If the set of conditions for a production rule P is a proper subset of the conditions for production rule Q, then rule Q will fire first. Rule Q is more specific than P because of

its additional conditions, hence the interpreter should address the more detailed prior to the more general case.

If the working memory elements which satisfy the conditions for production rules P and Q differ only in that the elements for P were created or modified more recently than those for Q, then rule P will fire. Thus, expansion is depth first in that once a path is followed, it will be continued as far as it can go before branching out.

This strategy introduces order into a potentially chaotic situation. Obviously, it is necessary if the 'expert' is to have any control over the system. Knowledge of the mechanism at work allows programming of specific tasks though the control flow may be subtle or possibly convoluted.

## D. A SAMPLE PROGRAM

As with any language, learning its primitives and syntax is the major hurdle to successful programming. But our purpose is to determine the suitability of OPS7 for integration into ZOG. To this end, a small sample program will be reviewed. The particulars regarding items such as input and output syntax are not important to this example. If the reader has further interest in such matters, see [Ref. 7].

This program asks the user to input numbers and, when told to sort, will print out the numbers in ascending order. To preserve simplicity no error checking is performed.

TYPE DECIARATIONS

```
(type number                        ; number schema
      value = scalar )

(type task                          ; task schema
      type = scalar )
```

PRODUCTION RULES

```
(p readin                           ; A rule called READIN
   i(task type ¬= sort              ; Read as long as the input
    )                               ; is anything but the word
                                    ; 'sort'.  NO ERHOR CHECKING
                                    ; 'i' is a label.
-->
   (write "Input an Integer-->"
         [ / / * / ]                ; Input message
  )

  (modify i type =
     (index (accept) 1)             ; Read value into task type
  )
  (make number value =
        i:type )                    ; Make another INSTANCE of
                                    ; number using the same value )

(p sort                             ; A rule called SORT
     (task type = sort)             ; Task is now sort
                                    ; (as entered by user)
    j(number value ¬= sort)         ; Find a number j which
                                    ; is not = sort,
                                    ; ('j' is a label)
    -(number value ¬= sort          ; and there is NO value
                                    ; (including 'sort')
            value < j:value         ; which is smaller than j
     ) -->
     (write j:value
           [ / * ]                  ; Print out the smallest value
     )
     (remove j)     ; Remove this value from working memory )
```

INPUT DATA

```
(make task)         ; Create an instance of task
                    ; to start the system
```

The order of entry is important in OPS7; type declarations, rules, and then input data. Generally, the instances created by the input data establish the working memory elements required to start the system. The type declarations are easy enough to understand. Two types are declared, both of which hold scalar values. The INPUT DATA makes an instance of type task and assigns its field type the default scalar value of '?'. Placing this in working memory causes the first p rule, READIN, to fire. READIN assigns the input value to both number value and task type. This rule will continue to fire until the work 'sort' is typed and entered. Once this happens, p rule SORT will fire because the task type = sort and there exists (at least one) number value NOT equal to sort. The beauty of OPS7 is seen in this simple production: the interpreter has been instructed to search working memory until it find a value 'j' which is smaller than any other value (not equal to 'sort'). This smallest value is printed to the console and removed from working memory. SORT continues to fire until all number value instances, except value = sort, are removed from working memory. At this point there are no working memory elements which can satisfy the right hand sides of any p rules, so the system halts. Essentially, this is a program which will sort from one to many numbers (restricted by memory size) using only one rule!

## E. SUMMARY

This review shows that there is a structure in the creation of an OPS7 system which can be supported by a hierarchical environment such as ZOG. Specifically, subnets in ZOG can be the structures for storing type declarations, working memory (each frame containing a single working memory element), production rules, and the conflict set.

The job of the interpreter will be to know the location of these subnets and what to do with specific types of frames. The next chapter will discuss the design of the frames for each such subnet.

# V. FRAME SCHEMA DESIGNS

## A. INTRODUCTION

Now that there is an understanding of the frame structure in ZOG and the format of OPS7, the next step is to develop subnets which the new interactive interpreter will use, and to detail the basic design of the generic frame, or schema frame, for each subnet. A subnet is required for: 1) each system subnet used and maintained by the interpreter, and 2) each user subnet upon which the user can develop his programs. It follows that each subnet should have a unique frame schema so that the interpreter knows what to expect each time it refers to it. The unique schemas take advantage of the structure and syntax of OPS7 for writing programs and organizing subnets.

The first consideration for schema design is whether the information written on the frames should be frame text or frame options. Because each of these parts of a frame are implemented as selection pointers, it makes little difference to the system which one is used when trying to find information on the frame. If the frame item is to point to another frame, options are required. It is also preferable to have frames which contain only text, without selecting other frames. For these reasons both text and options are used.

The use of the frame determines which global pads are displayed. Those frames which are created by the user will have a standard set of global pads (table I). Those frames created, modified, and removed by the interpreter will contain a similar set except 'edit' will not be available.

**TABLE I**

**STANDARD GLOBAL PAD SET**

| | | |
|---|---|---|
| EDIT | - | RUN 'edit', THE ZOG EDITOR. |
| HELP | - | DISPLAY THE TOP FRAME OF ZOG USER'S GUIDE IN THE OTHER WINDOW. |
| BACK | - | BACK UP ONE FRAME IN THE BACK-UP STACK. |
| NEXT | - | DISPLAY THE NEXT OPTION FROM THE SELECTION FRAME. |
| PREV | - | DISPLAY THE PREVIOUS OPTION FROM THE SELECTION FRAME. |
| TOP | - | DISPLAY THE TOP OF THE NET (FOR THE PARTICULAR MACHINE). |
| GOTO | - | GO TO SPECIFIED FRAMEID. SYSTEM WILL PROMPT. |
| UTIL | - | DISPLAY THE TOP FRAME OF SUBNET ZOG. SHOWS AVAILABLE AGENTS AND ACTIONS. |
| DISP | - | REDISPLAY THE CURRENT FRAME. |
| INFO | - | DISPLAY FRAME MAINTENANCE INFORMATION |
| WIN | - | MAKE THE OTHER WINDOW THE CURRENT WINDOW. |
| JUMP | - | PUT THE CURRENT WINDOW FRAME IN THE OTHER WINDOW AND CHANGE WINDOWS. |

The subnets for this proposed OPS7 system contain all the information that the interpreter requires for proper execution. Each have unique characteristics causing the appropriate results when it interacts with the interpreter. The requirement for subnets can be divided into two types: those created by the user and those created by the interpreter. The user subnets are the working areas in which declarations, rules, and actions are written by the programmer. Characteristic of these subnets are frames which allow editing for the purposes of writing OPS7 programs. The interpreter, or system, nets are similar in form to the user nets, except editing of the frames is denied. This is accomplished by write protecting the frames and by removing the 'edit' global pad from the frames. This prevents the user from circumventing consistency checks done by the system. System subnets are required for type declarations, production rules, working memory, and the conflict set. What follows are specific designs for the schema frames for each subnet.

## B. THE USER SUBNETS

When the user elects to write OPS7 programs in ZOG, the first frame presented to him is the environment frame for the agent. Agents typically require one or more user-specified parameters, such as subnet name, or output file name, in order to run. Environment frames were created to provide a means of passing this information to the agent. These frames use their options as 'slots' that hold this input data. The slot editor is used in conjunction with these frames to provide the user a simple method of inserting the desired information.

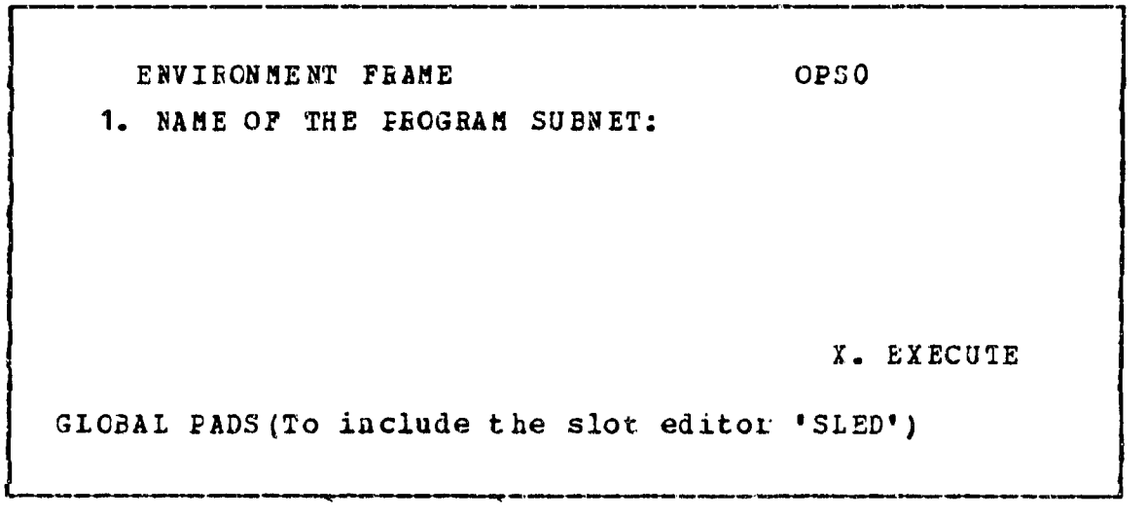


**Figure 5.1 OPS7 Environment Frame.**

[Ref. 8]. In this instance, the user selects the slot editor and, following a prompt for the subnet name, fills in the name of the subnet he wishes to use. Error checking done by the slot editor prevents entering an invalid subnet name. The exection local pad on the environment frame causes the agent to begin execution on the given subnet.

If the subnet already exists, the agent presents the top frame in the subnet in the current window and the user proceeds as desired. If it is not present, the agent creates a new subnet using the name from the input slot and copying the PROGRAM0 frame as a schema (see figure 5.2). The system then creates the subnets for types, rules, and actions under the respective options, using the following naming convention. The subnet names include the subnet function (type, rule, or action) appended to the end of the subnet name from the input slot, not to exceed 12 characters. For example, for a program name of AIR, the subnets are called AIRTYPE, AIRRULE, and AIRACTION. If need be, letters are truncated from the input slot subnet name. The development of these subnets is discussed later. The PROGRAM0 frame schema contains the standard set of global pads and a set of six local pads: Load, Run, Halt/Continue, Working Memory, Conflict Set, and Error Msgs. The Load pad is selected once the program has been entered. It tells the interpreter to evaluate the program statements for syntax and type errors. The Run pad explicitly tells the system to commence evaluation and execution of the PROGRAM frame. The Halt/Continue pad allows the user to arbitrarily stop a running OPS7 program in order to go to other frames and analyze program actions. The Working Memory pad is a link to the existing working memory subnet. The Conflict Set pad links this frame to the conflict set subnet. The Error Msgs pad is the link to a frame which contains the text of the system error messages. Having these local pads makes the OPS7 actions, (wm) and (cs), obsolete as debugging commands. The top frame organizes the program into these specific subnets to make program creation and debugging easier for the programmer. Note that all schema figures may also include the syntax for <u>possible</u> entries into the frame.

On the top frames of each of the three subordinate subnets is where the programmer writes type declarations, production rules, and actions. Each entry is a single option on the frame. In the case of type declarations and p rules, only the first line of the declaration appears on the appropriate frame. For single actions not appearing as p rule right-hand-sides, the entire action statement is entered. For types and p rules, additional frames must be created to contain the body of these parts of the program.

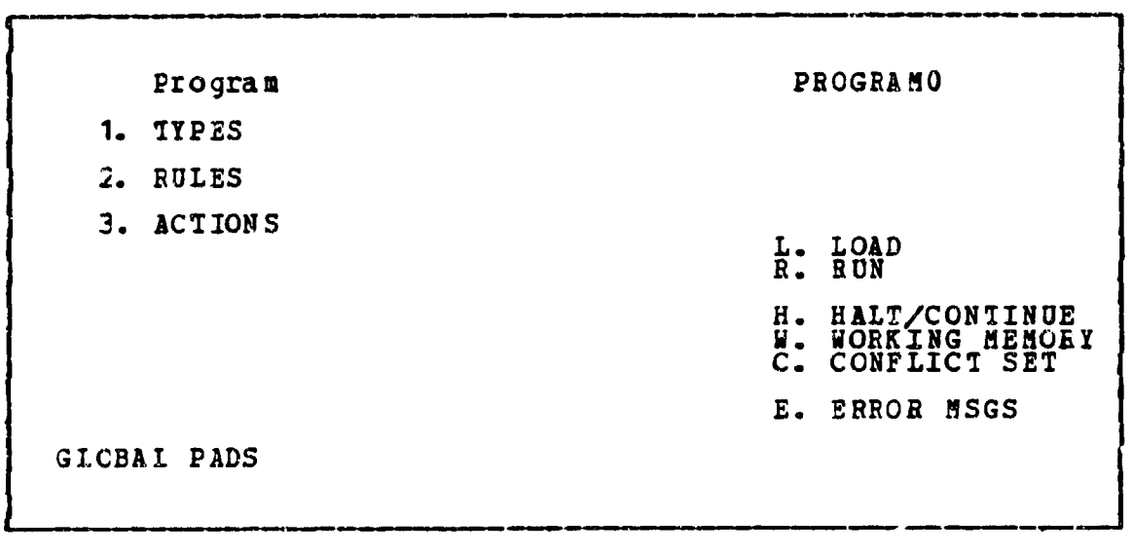


Figure 5.2 Program Schema Frame.

## 1. Type Declarations

The schema for the type subnet is a frame with the standard global pads and two local pads (Figure 5.3). This top frame is created by the agent and is linked to the top frame in the user subnet. The subnet name for this subnet comes from appending the word "types" on to the first seven letters of the user program name. The local pad, More, directs the user and the interpreter to additional

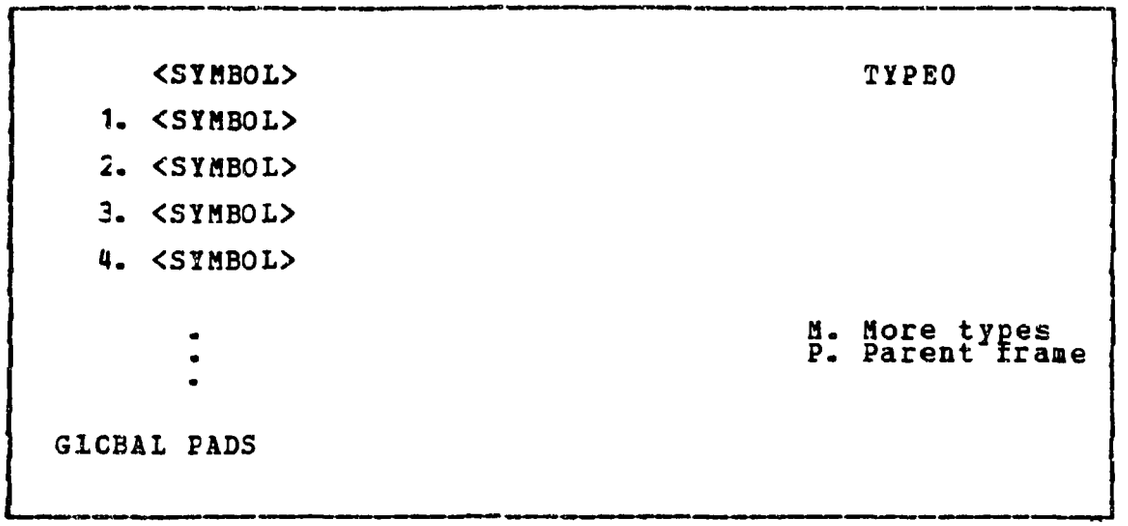


**Figure 5.3 Type Schema Frame.**

type declarations should more than nine be needed ( there are a maximum of nine options per normal frame). While the More pad is not the most efficient way to traverse a list of items, this system should not have to support a program with more than two frames worth of types. The issue of program size is addressed later. The parent pad directs the user to the current frame's parent.

To create type declarations the user selects the TYPES options on the top PROGRAM frame. This selection leads to the top of the type subnet. The programmer then selects edit and enter the first type as an option. Once out of the editor, the desired type is selected and the ELEMENT0 schema is explicitly requested. This frame is created and the editor is automatically entered (see figure 5.4). The body of the type declaration is entered as text, in accordance with the OPS7 syntax, and saved. When the interpreter encounters the TYPE option on the top PROGRAM frame, as it process the frames beneth it, it enters the types found in a subnet called GlobalNet; This process is explained in section C.1.

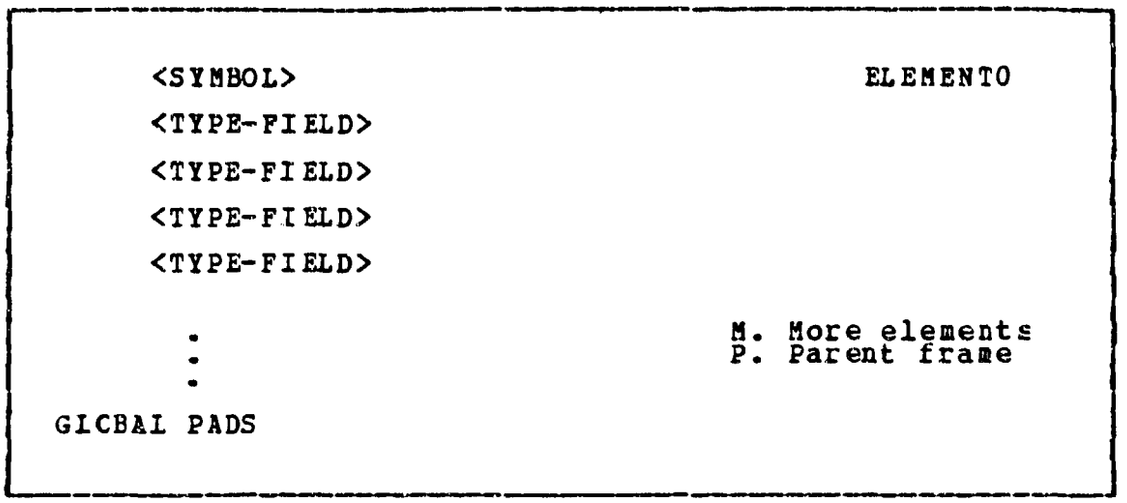


**Figure 5.4 Type Element Schema Frame.**

2. P Rules

The body of p rules are entered like types using a schema similar to TYPE0, except the More pad leads to 'More rules.' This schema is called RULE0. The tree below this frame differs from the types tree because at least two frames are needed to contain a single p rule: one for conditions and one for actions. The schema frame for conditions contains the standard global pads and the same local pads as TYPE0. The use of a More pad is considered sufficient here because in most cases rules can be expected to have fewer than twenty-one conditions (there area maximum of twenty-one line of text per normal frame). The conditions are entered as text on these frames. The bottom of the condition frame contains an option '-->', which points to the actions for the given p rule. This option does not appear if there are more conditions on another frame. Figures 5.5 and 5.6 illustrate these schema frames.

The action schema frame is a frame like ELEMENTO, with the standard set of global pads and the local pads, More and Parent. Actions which appear on the right-hand-side of production rules may also stand alone as commands in OPS7. The standard use for these kinds of actions is to create some initial state in working memory so the program starts when run is selected. This subnet is modified by selecting the ACTION option at the top of the user subnet, selecting 'edit' on the frame, and entering the action as text.

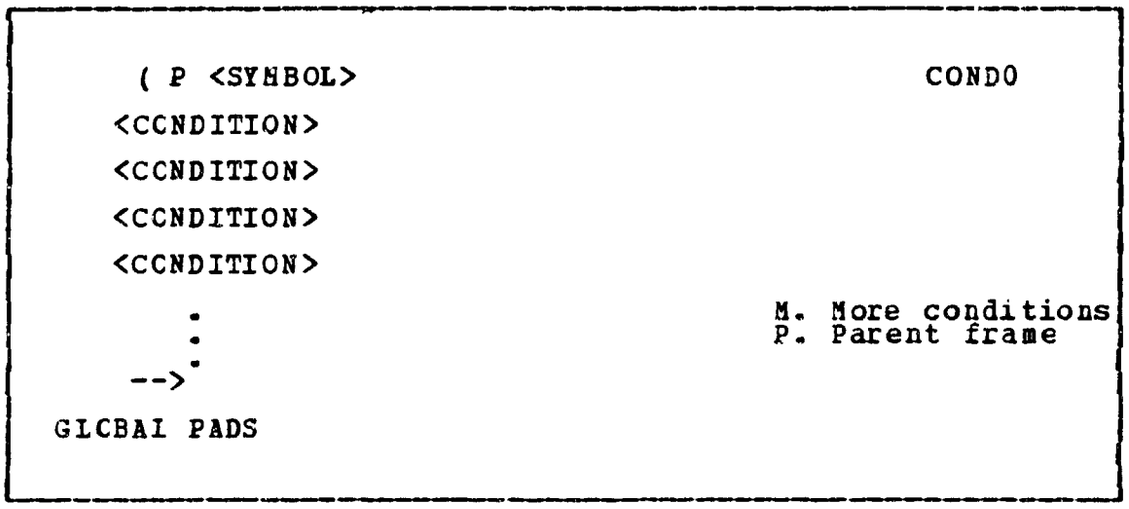


**Figure 5.5 P Rule Condition Schema Frame.**

## C. SYSTEM SUBNETS

The system subnets are those created and maintained by the interpreter. The subnets required by the interpreter are for global variables, working memory elements, and the conflict set. They are called GlobalNet, WM, and CS, respectively. These subnets are created the first time the interpreter is called to load a program. Subsequently, any

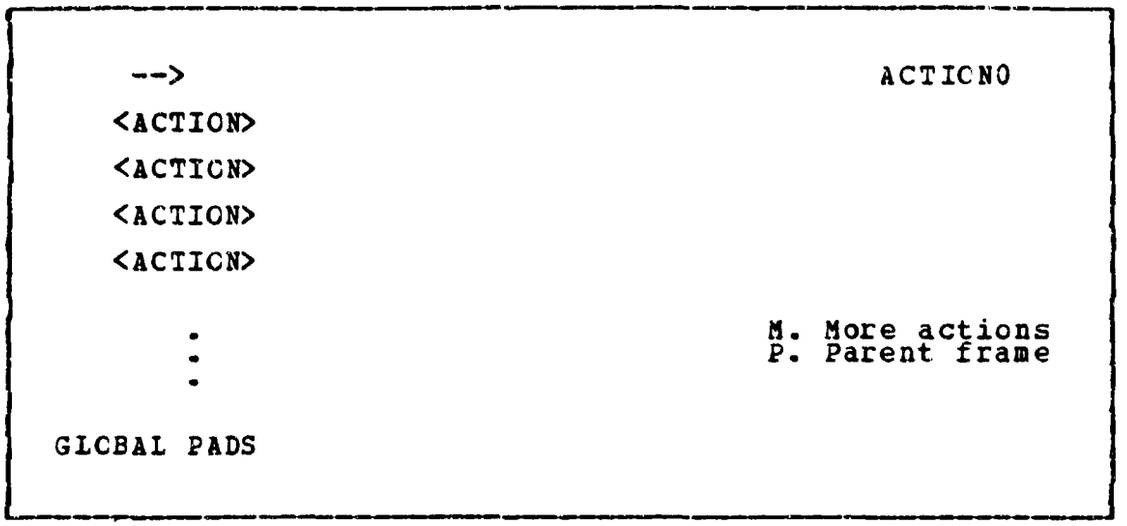


**Figure 5.6 P Rule Action Schema Frame.**

time another program is loaded, they are cleared out so the system can start from scratch. The subnet names for these nets are not concatenated with the name of the user program subnet because they are independent of any particular program. As previously mentioned, security is maintained in these subnets by DENYING the user the ability to edit system subnet frames.

1. Global Subnet

As the interpreter is processing, each time a type declaration is found it is inserted into the GlobalNet. This is the system's reference mechanism whenever it is creating an instance of a declared type for working memory. This name comes from the fact that all variables is OPS7 are global [Ref. 6].

Adding an entry into this subnet is a two step process. First, the type name (OPS7 syntax for this is <SYMBOL>) must be written as an option in the top frame of the subnet. Figure 5.7 shows the top frame in the GlobalNet subnet.

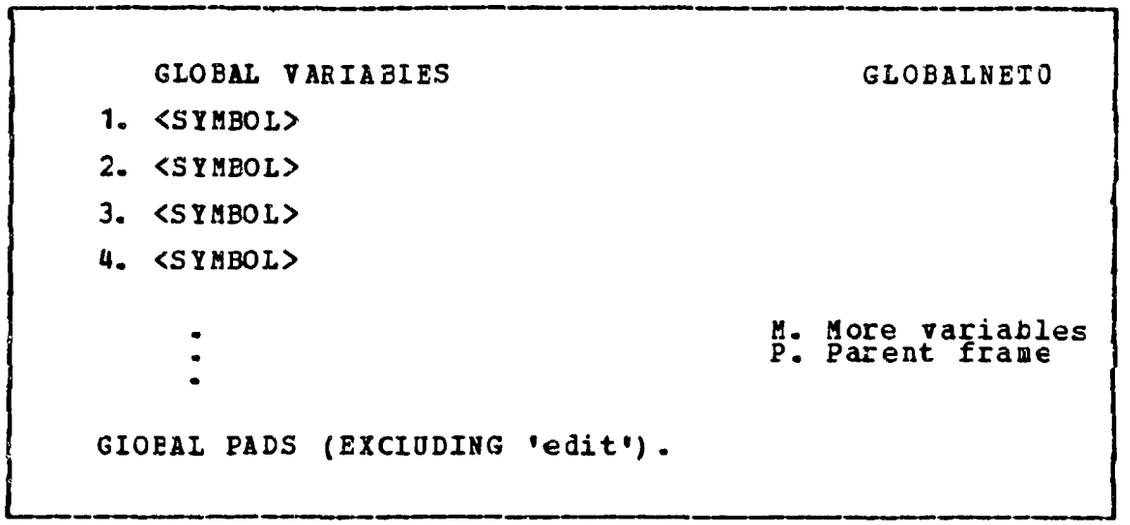


**Figure 5.7 GlobalNet Schema Frame.**

The second step is the creation of the frame which has the actual declaration. This frame contains infcrmation as text. To insure security of system nets, a copy of the type elements frame from the user subnet, TYPES, must be copied into this frame. Simply establishing a link between the GlobalNet and the TYPES subnet would allow the user to edit frames used by the system. The system does not create these frames until they have been found syntactically correct by the interpreter.

2. Working Memory

The working memory subnet is used by the system to hold working memory elements, which are specific instances of the previously declared types. The top frame in this subnet is similar to that of the GlobalNet except its subnet name is WM (refer to figure 5.7). The subordinate frames in this subnet use ELEMENT0 frames as the schema. When the interpreter encounters the function MAKE (implicitly or explicitly), it creates an option on the top WM frame and an

instance of that type from GlobalNet is copied into the Working Memory subnet.

This subnet's element frame differsfrom the GlobalNet's type schema in that the values assigned to the various fields of the element are included in the text. Each value is appended to the end of the text containing the type-field. As seen in figure 5.8, the character

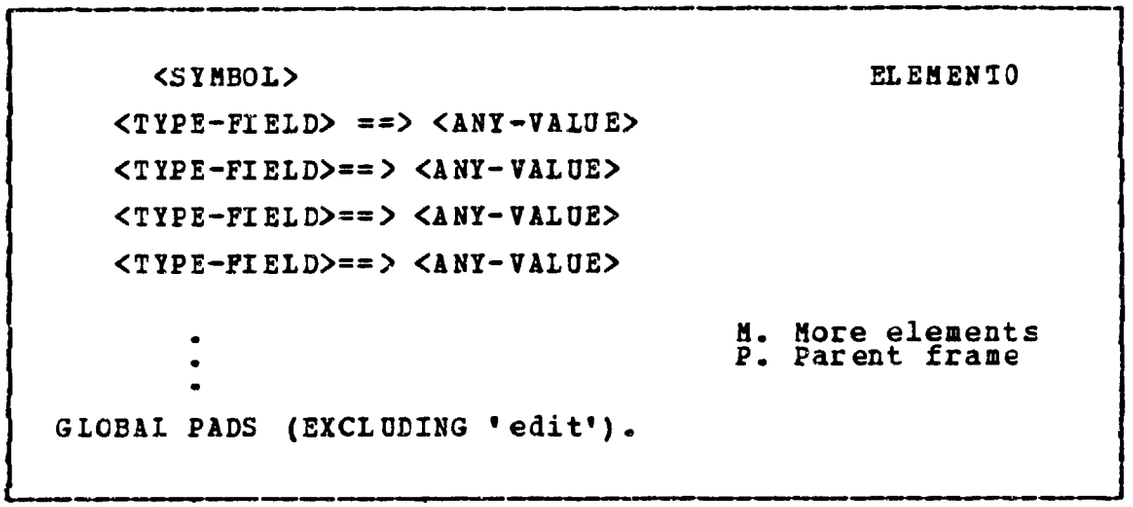


**Figure 5.8 Working Memory Element Schema Frame.**

string '==>' separates the declaration from the value. The creation of the working memory element requires writing the element name ( <symbol> ) on the top frame in the WM subnet, copying the type information frame from the GlobalNet into its element frame, and writing the explicit values (or the defaults) assigned by Make.

A potential problem arises when one considers how many working memory elements might be created during a program run. It is difficult to estimate this because it depends not only on the nature of the program, but also on the different ways a program can be executed. What is known, is that the conflict set must have a unique way of

identifying each element in working memory. The solution to this is to use the selection number of the working memory element option appended to the number part of the frameid to create a unique identification number. When a working memory element satisfies a p rule condition, the above number is passed to the conflict set subnet.

3. Conflict Set

The OPS7 conflict set contains the p rule name(s) and the ID numbers of the working memory element(s) which satisfy conditions of the particular rules. Using the example from the previous chapter, if the p rule SORT had its two conditions satisfied by elements 1 and 9 from the working memory, then the response to the OPS7 action (cs) would be to display the conflict set 'SORT [1 9].' The conflict set may contain zero, or more elements.

```
    CONFLICT SET                                        CS0
  <SYMBOL>  [ELEMENT ID NUMBERS]
  <SYMBOL>  [ELEMENT ID NUMBERS]
  <SYMBOL>  [ELEMENT ID NUMBERS]
  <SYMBOL>  [ELEMENT ID NUMBERS]


       .                                    M. More
       .                                    P. Parent frame
       .
 GLOBAL PADS (EXCLUDING 'edit').
```

Figure 5.9 Conflict Set Schema Frame.

In the ZOG implementation of OPS7 the conflict set is kept on frames in a specificly created subnet. The schema frame for this subnet is a frame with the standard global pads without 'edit', and local pads for Parent and More. The conflict set information is written as text. Figure 5.9 illustrates this schema. This subnet is used once the run command is executed at the top of the user subnet. Every time the recognize and act cycle completes, the information on this frame is updated because the p rule which just fired must be removed. Remaining unfired rules are also deleted if the change in Working Memory caused by the firing of the previous rule invalidated a rule in the conflict set.

## D. SUMMARY

The subnets defined here are the mechanisms through which a user can use OPS7 in the interactive ZOG environment. Being able tc do this on ZOG frames takes advantage of ZCG's hierarchical organization and fast retrieval of information. But there is still an important piece of the puzzle missing. The OPS7 interpreter envisioned must be called from within ZOG to create, execute and most importantly, debug programs. Hopefully, these subnets establish an intuitive framework within ZOG for the programmer and the interpreter.

# VI. AN INTERPRETER

With the foundation laid by previous chapters, this chapter outlines the characteristics of the interpreter to be used in this system. This is done by reviewing three things: 1) the nature of the design for OPS7, 2) what features the interpreter will have, and 3) issues which will directly affect the practicality of the implementation.

## A. DESIGN NATURE

In trying to decide just what the interpreter for this system should look like, two distinct choices were apparent. First, the system could be an interface between the existing ZOG system and the current OPS7 interpreter. Essentially, this would mean creating a layer of software between ZOG and OPS7 for the purpose of formatting data into a useable form for each system. Although the appearance of OPS7 in ZOG would be hierarchical and more interactive, it really would be the old interpreter in disguise. This approach sidesteps the entire issue of designing a new tool for the environment. The second choice is to integrate the features of OPS7 into ZOG itself. To do this OPS7 must be able to communicate to the user via ZOG mechanisms while continuing to evaluate and execute programs correctly.

The decision of which option is preferable is based on a number of engineering issues such as the time constraints on the design project and the compatibility of the OPS7 system with ZOG. The determination of which implementation approach is easier, is not trivial due to the complexity of ZOG and OPS7. If the project were time sensitive, the method which would get a system up and running the fastest

is the obvious choice. Comparing this research with other projects may provide some insight into this decision. Regarding compatibility, the two systems are currently written in the same programming language, and OPS7 programs have an inherent hierarchical structure like the organization of data in ZOG. While the use of the same programming language does not imply compatibility, the similarity in organization of the two systems does.

It is because of the desire to implement OPS7 under ZOG's control and the compatibility of system organization that the latter direction is chosen.

## B. AGENT FEATURES

The interpreter, which currently comprises 14 modules, will be integrated into ZOG as an agent. Agents are basically processes within ZOG which know about ZOG structures. Typically agents operate on subnets of frames, or portions thereof [Ref. 8]. Their main purpose is to extend the functionality of ZOG. Agents differ from system utilities in that the latter are components of ZOG. The former are programs that are separate yet called from within ZOG [Ref. 5].

There are many features which will be a part of the design of this interpreter, and will perform implicit tasks, such as the creation of the Working Memory and Conflict Set subnets. The user will interact with the explicit features for the creation and debugging of programs. To illustrate how the agent will work, a sample programming session will be discussed.

In ZOG, the programmer will select the OPS7 interpreter by calling up the net utilities frame and choosing to run the OPS7 agent. If the subnet name given to the environment frame is not found, the agent will create a total of seven

subnets. For the user, the top PROGRAM frame with the three options is created. Each option points to the respective subnets for types, rules, and actions. When the system initialization is complete, the top frame in the user subnet will be in the current window. The programmer may now enter statements on the user frames by selecting the appropriate options. As described in chapter 5, only parts of the statements may appear on the top user frames for types and rules. The programmer may choose to first enter all the required syntax for these frames in a top-down fashion, or to enter the statements and its body (on a connecting frame) before proceeding tc the next option on the parent frame. The latter approach is called depth first. The top-down method is faster because the top frame in the subnet will only have to be opened and closed once.

When the program has been entered the user must return to the top frame and select the Load local pad. This will cause the interpreter to traverse the program subnet performing type checking, creating the GlobalNet, WM, and CS subnets, and performing any actions present. At this time OPS7 will also put the production rules into what it calls production memory. Essentially, this is a translation of the language syntax into a more compact form for use by the production system part of the interpreter. Any errors detected during the load phase, will be announced to the user's display on the current frame and written to an error message frame. This frame is found by returning to the top frame in the user subnet and selecting the local pad E. The errors will be options on the Error Msgs frame; these options will be linked backed to the frame in the user subnet which contains the error.

Suppose the program has been correctly entered and loaded. Now the Run local pad is selected. This action causes the recognize and act cycle to commence.

Communication with the user is accomplished through the user display of the current window (which is at the top of the user subnet). Although the mechanics of this process are transparent to the user, the interpreter searches through working memory trying to find elements to match the left-hand-sides of production rules.

As the production system is firing its rules, three main things are happening. First, the conflict set subnet is continually expanding and contracting as conditions are satisfied. Closely connected with this is the second activity, the creation, modification, and removal of items from the working memory subnet. Most important is the third activity: the system I/O with the user. This takes place in the user display, and allows a somewhat limited method of communication with the program.

While the production system is running, it would be advantageous to allow the user to look at a system subnet, such as WM or CS, in order to follow what the program is doing to. A feature implemented specifically for AIRPLAN, called incremental display, would prove useful in implementing this capability. Incremental display has ZOG update the visual representation of a frame, which is displayed in one of the ZOG windows, whenever the physical information for that frame, in secondary storage, has changed. Implementing this while OPS7 is running could prove to be impractical because a software level interrupt would be required to tell ZOG to update the displayed frame.

In the event that the program has a semantic error which causes, for example, the program to terminate prematurely, the user may want to survey what the system was using for working memory or what was contained in its last conflict set. This is done by returning to the top of the program subnet (if not already there) and selecting the appropriate local pad. This feature has the potential to greatly

enhance the process of debugging, because while the program listing is in the bottom display window, the upper window can be used to traverse the desired subnet for debugging. Take the situation above. To find out why a program has halted one may want to view the condition frame of a p rule while viewing the contents of particular working memory elements. The next and previous global pads can be extremely helpful in this situation by allowing the programmer to move from condition frame to condition frame of different p rules without returning to the parent frame holding the selections. Similarly, elements of working memory may be viewed.

As previously mentioned, the programmer will be denied the ability to edit system subnets via global pads. But there does exist an alternate method of entering the editor on the current frame. If the user logged into ZOG is the frame owner, the editor may be selected by typing control-d, followed by e. One would only want to do this to manipulate memory elements that might be hindering a program's development. The bug creating the specific problem could be overlooked in order to allow the program run to completion. The freedom to do this would be restricted by having the agent make a special owner assignment when creating the system subnets. The password to log in as this special user would be limited to the OPS7 implementor(s). It is their responsibility to realize the unpredictable results which may occur if illegal modifications are made to subnets maintained by the system.

## C. IMPLEMENTATION ISSUES

Because the interpreter and the interactive environment are real world systems, it is appropriate to discuss some known issues which must eventually be dealt with if such a

project is to ever be implemented. While this section attempts to bring to light implementation questions and suggest possible solutions, in no way is the domain of solutions limited to the author's knowledge nor the limitations of the current systems. While some of the issues may seem prohibitive, the impact of future technological capabilities can not be dismissed.

1. Writing to PCS files

An initial question is how an OPS7 program in the ZOG system will be saved into a PERQ Operating System (POS) file. The capability to write the information from frames into operating system files currently exists in ZOG. The agents designed to do this must be modified to read the program from the subnet in the proper sequence. The ability to do this is necessary because programs may be smaller components of larger OPS7 programs too big for use in ZOG. This integration of modules is currently envisioned to be done outside the developement environment.

2. Program Size

Program size is an issue which impacts the design of every subnet in the proposed OPS7 implementation. When considering size, one should look at an existing expert system. AIRPLAN is the only one currently implemented in OPS7. In its present form, it uses about 200 p rules. The goal of this research was to create a development environment for small programs or parts of larger programs. Hence a program of AIRPLAN's size and complexity was not planned to run in this environment. This decision may seem arbitrary, but the author felt that if this system could be implemented with this limit, expanding it to support full OPS7 programs could be dealt with later. Specifically, the author envisioned the ability to support programs about one-fourth the size of AIRPLAN.

### 3. PERQ Hardware Limitations

Another issue is the implementation of ZOG and OPS7 together on the PERQ. The PERQ can support 32,000 16-bit words of global data (in Pascal, under the PERQ Operating System). The current version of ZOG uses about 24,000 global words. OPS7 requires 23,000 global words. One can reduce this number by converting large structures (frames on down to individual strings) from static variables (currently managed by ZOG) to dynamically allocated structures using the Pascal NEW call. This still requires the use of a 32-bit pointer to the structures in the global word area of memory, and the pointer will have to be dereferenced every time the structure is referenced. The dereferencing will add some extra time overhead. It is possible to combine ZOG and OPS7 on the PERQ using this method.

A more pressing problem is the amount of primary memory available. When ZOG was first put on the PERQ, the implementors tried to include a Pascal Compiler. This resulted in the system swapping so much that it was functionally brought to a standstill. The simple solution is to hope for the availability of a larger memory board for the PERQ. Currently, an upgrade from 1 Megabyte to 2 Megabyte memory boards is being investigated onboard the Carl Vinson. Without such a change, the only option available would be to include in ZOG a subset of OPS7. The majority of the globals for OPS7 are concentrated in two modules. This implies that some sort of reduction of the standard OPS7 may be possible [Ref. 9].

### 4. Benefits of OPS7 in ZOG

One might ask what is the benefit of having OPS7 in ZOG in terms of the time required to simply type in the subject program. In other words, will the programmer spend

more time trying to enter a program in ZOG than he otherwise would typing it into a text file. Based on experience with AIRPLAN, the following can be said. Unquestionably, for an inexperienced user, the use of a text editor is definitely preferred over ZOG. This is because the user would be spending most of the time trying to understand how ZOG works, rather than concentrating on program creation. Once the user becomes more familiar with ZOG, the time required to create an OPS7 program should decrease dramatically.

ZOG is designed to be an intuitive, easiy to learn, human-computer interface, but in reality, it takes hours of use before the user can adroitly construct trees and edit frames. In this implementation, the ZOG environment would be used to add organization to OPS7 but not make it easier to type in programs. Hopefully, the benefits of having OPS7 in ZOG will more than compensate for the increased overhead required to use ZOG frames.

### 5. System Execution Time

The time required to run this system is interesting to analyze. Fairly good numbers exist for determining the time it takes to read a frame from disk memory into the ZOG Pascal record structure. For a local frame it takes 0.5 seconds to read a small frame; 1.2 seconds are required for a remote frame. The time to modify a frame (an Open followed by a Close) is approximately double the read time [Ref. 9]. The time required to locally create a small frame is estimated to be approximately 2 seconds. The following is an estimation of the time required to load and run the number scrt program in Chapter IV.

This program has two type declarations. The interpreter will open the type frame to find the first type name and the location of the element declaration frame. It will then open the GlobalNet and write in the type name as the

first option, followed by copying the declaration frame into a newly created frame pointed to by the GlobalNet option. This process requires the opening and closing of a minimum of three frames and the creation of another. This must be done for every type declaration. Therefore, about 5 seconds will be required to load each type, or a total of 10 seconds for the program.

Reading the p rules into the system production memory requires reading the rules frame for the individual rules, and reading both the condition and action frames. For each rule, a minimum of three frames must be opened and closed; this will take about six seconds. Allowing time for the reading of the rule into memory means each p rule will required about seven seconds. The two p rules in the example will require 14 seconds to load.

The single action in this program is read from its frame and the 'make' directs the interpreter to create a working memory element. This process requires the opening and closing of two frames, the reading of another, and the creation and modification of a third. It is estimated that this will require 4.5 seconds (3.5 of which is required to create a working memory element). The total time required to load this program is about 28.5 seconds.

Program run time is determined by following the action of the program. When run is selected, the system will attempt to find matches for the p rule conditions in working memory. This process requires the system to read every working memory element for each p rule. To do this the top frame in the WM subnet is read for the element IDs and the pointer to the element values. In other words, on every recognize and act cycle at least two frames must be read per working memory element. Then, each time a match is found for the left-hand-side of a p rule, the CS subnet must be opened so the new member of the conflict set may be

entered. It will take 3 seconds to fire the first rule. As working memory increases the time required to read working memory will increase linearly. The time to create the CS subnet will depend on the nature of working memory.

The readin rule will continue to fire until the word 'sort' is entered by the user. The time between user inputs is devoted to making a new memory element, modifying another, and determining the conflict set for the next recognize and act cycle. This will take about 1 second more for every new number added. Entering five numbers to be sorted, plus the word sort will take at least 40 seconds. The processes to perform the sort will take about the same length of time.

All told, the loading and running of this small program, assuming the subnets are local, will take at least 110 seconds. This is due mainly to the time required to create, open, and close so many frames. The same load and run process on the existing OPS7 system takes only about 10 seconds! This is a tenfold decrease in performance. Extrapolating these performance figures to a program containing 25 rules may make the new system intolerably slow.

A method to increase performance is not clear because the bottleneck lies in the overhead for reading frames from disk storage. Like the previous issue of main memory size, the only solution to increased speed may be found in improved CPU technology. In any case, the benefit of having OPS7 in ZOG will have to be weighed against this degredation in execution performance.

## D. SUMMARY

The issues addressed here are by no means all that need be considered, but they do represent the real world considerations that must be faced for projects in general, and this research in particular. Should the schemas proposed by this research be implemented, these issues must be resolved if it to have any impact on interactive programming.

# VII. CONCLUSIONS AND RECOMMENDATIONS

## A. CONCLUSIONS

This research was an investigation into the design of an interactive programming environment. The requirements for such an environment were initially studied. This research showed that the environment should (1) provide tools specific to the supported language, (2) use the underlying structure of the language in designing the environment, (3) support incremental program development, and (4) support a high bandwidth of communciations between the user and the environment.

In analyzing ZOG, one finds that it conforms nicely to this paradigm, except it does not have any specific tools to support the desired expert system language, OPS7. In order to design these tools a study of the code for both ZOG and OPS7 was undertaken. It should be noted that the time to do this study took much longer than expected because of the size of the two systems and lack of instructional documentation of the system code. The result of this study was the design of a reasonable framework for the writing of OPS7 programs in ZOG.

The design of the the subnet framework is the first step in the creation of the programming environment. During the actual implementation, issues concerning hardware limitations, the speed with which ZOG can run OPS7, and the time saved by developing OPS7 programs would have to be dealt with. These issues have been analyzed and solutions suggested.

## B. RECOMMENDATIONS

The experience of working with these two systems will undoubtedly pay dividends in the future. The work experience gained by studying both ZOG and OPS7 have instilled in the author an appreciation for the effort, both in research and manpower, required to design and implement human-computer interfaces and new programming languages. Also, the author will be leaving Monterey to work with the implementation of these systems onboard USS Carl Vinson. Additional formal instruction in these systems will be forthcoming, but the time spent on this research has laid an important foundation for continued work in this field.

From this experience, it appears that trying to learn about a complex software system in a benign environment is difficult, at best. The learning environment must be similar to the real world environment, and have support from personnel, as well as documentation. Personnel must be able to provide to the student the benefits of their experience with the system. Further, the available documentation must extend beyond system definitions and source code to be of any tangible value.

The time required to understand large, complex software systems is difficult to estimate. The size of the system will have the greatest impact on the learing process. The next factor is the structure of the software. If the system is written in a structured programming language, some structure is inherent. Beyond this, the different modules of program code must be logically interrelated. Finally, the documentation available must extend to instruction on the design conventions used and implementations made during system development. This kind of documentation will help the user understand the overall design approach and prevent him from repeating mistakes made earlier in the system

development. Ultimately, the system size is the key. It may be well documented, have discrete, well defined modules, and be supported by many knowledgeable users, but with a large system, more time is required to understand enough so that the user can comfortably work with the system.

In future research in the area of interactive environments it is strongly recommended that implementation work in systems of this scale include experience tour type training in the subject system. The time required to bring the student up to the level of understanding required to accomplish this kind of work, is otherwise not available. In this instance, on-site facilities and technical expertise were available, but not to the degree sufficient to support further implementation work.

## APPENDIX A
### FRAME STRUCTURE SOURCE CODE

```
(GENERAL TYPE DEFINITIONS)

 *** NOTE: The symbol @ is used as the pointer label.

type  int         = integer;
      Pos Typ     = int;
      string15    = string[15];
      zstring     = string [255];
      SidTyp      = string15; {Subnet ID}
      FidTyp      = string15; {Frame ID}
      UsrIdTyp    = string15; {User ID}

{protection type}

    ProtTyp  = int;

{FRAME STRUCTURES}

{Short string structures}

type
    Fs15PTyp = @Fs15typ; {Pointer to frame string 15 }
    Fs15Typ  = record
     text:      string15; {a line of text }
     prevstr:   Fs15PTyp; {Pointer to the previous string }
     nextstr:   Fs15PTyp; {Pointer to the next string }
    end; {Fs15PType record}

type
    UsrIdPTyp = Fs15PTyp; {List of user ID's }

{String structure} type
    FsPTyp = @FsTyp; {Pointer to frame string }
    FsTyp  = record
```

```
    text:      string; {a line of text }
    prevstr:   FsPTyp; {Pointer to the previous string }
    nextstr:   FsPTyp; {Pointer to the next string }
  end; {FsPType record}

{Selection structure}

type
  SelPTyp = @SelTyp; {Pointer to selection}
  SelTyp  = record
    k:         char; {Selection character}
    nf:        FidTyp; {Next frame ID}
    text:      FsTyp; {Item of text }
    row:       PosTyp; {Item row position in the frame }
    column:    PosTyp; {Item column position}
    l0:        PosTyp; {Item minimum row position}
    c0:        PosTyp; {Item minimum column position}
    l1:        PosTyp; {Item maximum row position}
    c1:        PosTyp; {Item maximum column position}
    action:    FsPTyp; {Item action  }
    expand:    FsPTyp; {Expansion area }
    prevsel:   SelPTyp; {Previous selection}
    nextsel:   SelPTyp; {Next selection }
  end; {SelTyp record}

{Whole frame structure}

type
  FPTyp = @FTyp; {Pointer to frame}
  FTyp  = record
    nextfr:    FPTyp; {Next frame (save list only)}
    frameid:   FidTyp; {Frame ID     }
    owners:    UsrIdTyp; {List fo frame owners}
    crdate:    long; {creation date (longer integer) }
    modifier:  UsrIdTyp; {modifier     }
    moddate:   long; {modification date }
    modtime:   long; {modification time }
```

```
      version:  int; {version number }
      prot:     ProtTyp; {frame protection}
      AgCrBit:  boolean; {agent created indicator }
      AgModBit: boolean; {agent modified indicator}
      title:    SelPTyp; {title info   }
      text:     SelPTyp; {text info    }
      options:  SelPTyp; {options lists}
      lpads:    SelPTyp; {local pad list }
      gpads:    FidTyp; {global pad frame}
      comment:  FsPTyp; {frame comment}
      accessor: Fs15PTyp; {frame accessor list}
    end; {FTyp record}

{Frame header structure}


type
    FHPTyp = ^FHTyp; {Pointer to frame header}
    FHTyp  = record
      nextfr:   FHPTyp; {next frame header (save list only) }
      frameid:  FidTyp; {Frame ID          }
      owners:   UsrIdTyp; {List fo frame owners}
      crdate:   long; {creation date (longer integer) }
      modifier: UsrIdTyp; {modifier          }
      moddate:  long; {modification date}
      modtime:  long; {modification time}
      version:  int; {version number   }
      prot:     ProtTyp; {frame protection }
      AgCrBit:  boolean; {agent created indicator}
      AgModBit: boolean; {agent modified indicator}
    end; {FHTyp record}
```

# APPENDIX B
## OPS7 BNF SYNTAX

This is the BNF syntax for OPS7. It is included in this document for the convenience of the reader. It was extracted in total from [Ref. 7]. The symbol '¬' is used for relation negation. The non-terminal <symbol> stands for any name or label. The symbol ... means repeat the PRECEDING item any number of times.

```
<type>              ::=  ( type <symbol> <type-field>... )

<type-field>        ::=  <symbol> = scalar
                    ::=  <symbol> = set : <integer>
                    ::=  <symbol> = vector : <integer>

<rule>              ::=  (p <symbol> <condition>... -->
                                          <action>... )

<condition>         ::=  <pattern>
                    ::=  -<pattern>
                    ::=  <symbol> <pattern>

<pattern>           ::=  ( <symbol> <lhs-term>... )

<lhs-term>          ::=  <symbol> <relation> <lhs-value>
                    ::=  <symbol> : <integer> <relation>
                                              <lhs-value>

<lhs-value>         ::=  <scalar-constant>
                    ::=  { <scalar-constant>... }
                    ::=  [ <scalar-constant>... ]
                    ::=  <fldval>

<scalar-constant>   ::=  <symbol>
                    ::=  <integer>

<fldval>            ::=  <symbol> : <symbol>
                    ::=  <symbol> : <symbol> : <integer>

<relation>          ::=  <scalar-scalar>
                    ::=  <scalar-struct>
                    ::=  <struct-scalar>
                    ::=  <struct-struct>

<scalar-scalar>     ::=   = | ¬= | < | ¬< | > | ¬>
```

```
<scalar-struct>     ::=  in | ¬in

<struct-scalar>     ::=  has | ¬has

<struct-struct>     ::=   =
                    ::=   ¬=
                    ::=   intr
                    ::=   ¬intr
                    ::=   sub
                    ::=   ¬sub
                    ::=   sup
                    ::=   ¬sup

<action>            ::=  <wm-action>
                    ::=  <pm-action>
                    ::=  <io-action>
                    ::=  <variable-action>
                    ::=  <control-action>

<pm-action>         ::=  <rule>
                    ::=  <type>

<wm-action>         ::=  (make <symbol> <rhs-term>... )
                    ::=  (modify <scalar-constant>
                                        <rhs-term>... )
                    ::=  (remove <scalar-constant>... )
                    ::=  <reset>
                    ::=  <implicit-make>

<rhs-term>          ::=  <symbol> = <any-value>

<io-action>         ::=  (write <any-value> )
                    ::=  (write <any-value> <vector-value> )
                    ::=  (write <any-value> <vector-value>
                                            <scalar-value> )
                    ::=          (ifile          <scalar-value>
<scalar-value>)
                    ::=          (ofile          <scalar-value>
<scalar-value>)
                    ::=  (close <scalar-value> )
                    ::=  (load <scalar-value> )

<control-action>    ::=  (trace <scalar-value> )
                    ::=  (wme <scalar-constant>... )
                    ::=  (wm)
                    ::=  (cs)
                    ::=  (run)
                    ::=  ( run <scalar-value> )
                    ::=  (match <scalar-value> )

<variable-action>   ::=  ( let <symbol> = <any-value> )

<implicit-make>     ::=  ( <symbol> <rhs-term>... )

<any-value>         ::= [ <scalar-value>... ]
                    ::= { <scalar-value>... }
                    ::=  <scalar-constant>
```

```
                    ::=  <rhs-field>
                    ::=  <function>

<rhs-field>         ::=  <scalar-constant> : <symbol>
                    ::=  <scalar-constant> : <symbol>
                                           : <integer>

<function>          ::=  ( + <scalar-value> <scalar-value> )
                    ::=  ( - <scalar-value> <scalar-value> )
                    ::=  ( * <scalar-value> <scalar-value> )
                    ::=  ( / <scalar-value> <scalar-value> )
                    ::=  (   <scalar-value> <scalar-value> )
                    ::=  (gensym )
                    ::=  (genint )
                    ::=  (accept )
                    ::=  (accept <scalar-value> )
                    ::=  (accept <scalar-value>
                                 <scalar-value> )
                    ::=  (val <symbol> )
                    ::=  (append <vector-value>
                                 <vector-value> )
                    ::=  (index <vector-value>
                                <scalar-value> )
                    ::=  (union <set-value> <set-value> )
                    ::=  (intr <set-value> <set-value> )
                    ::=  (get <type-name> <scalar-value>
                                          <scalar-value> )

<type-name>         ::=  scalar | vector | set
```

## LIST OF REFERENCES


1. Newell, A., D.L. McCracken, G.G. Robertson, R.M. Akscyn. ZOG and the USS CARL VINSON. Carnegie-Mellon University Computer Science Research Review, 1980/81.

2. Sobel, A. Developing Airplan, Computer Science Department, Carnegie-Mellon University, 31 July 1983.

3. Barstow, D., H. Shrobe, E. Sandewall. Interactive Programming Environments, New York: McGraw-Hill, 1984.

4. MacLennan, B. J. Principles of Programming Languages: Design, Evaluation, and Implementation, New York: Holt, Rinehart and Winston, 1983.

5. Yoder, E., R. Askcyn. ZOG User's Guide, Computer Science Department, Carnegie-Mellon University, Version One, 6 July 1982.

6. Brownston, L. S. Elements of OPS7 Programming Style Carnegie-Mellon University, December, 1982.

7. Forgy, Charles L. Preliminary OPS7 Manual Charles L. Forgy, 20 July 1982.

8. ZOG System Operational Description Carnegie-Mellon University, December 23, 1983.

9. ARPANET MAIL, from D. McCracken, Carnegie-Mellon University, to D. Butler, 18 June 1984.

## INITIAL DISTRIBUTION LIST

| | | No. Copies |
|---|---|---|
| 1. | Defense Technical Information Center<br>Cameron Station<br>Alexandria, Virginia 22314 | 2 |
| 2. | Dudley Knox Library, Code 0142<br>Naval Postgraduate School<br>Monterey, California 93943 | 2 |
| 3. | Department Chairman, Code 52<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, California 93943 | 1 |
| 4. | Office of Research Administration<br>Code 012A<br>Naval Postgraduate School<br>Monterey, California 93943 | 1 |
| 5. | Computer Technologies Curricular Office<br>Code 37<br>Naval Postgraduate School<br>Monterey, California 93943 | 1 |
| 6. | Dr. Bruce J. MacLennan<br>Code 52Ml<br>Naval Postgraduate School<br>Monterey, CA 93943 | 1 |
| 7. | LCDR Paul S. Fischbeck, USN<br>Code 55Fb<br>Naval Postgraduate School<br>Monterey, Ca. 93943 | 1 |
| 8. | Dr. Don L. McCracken<br>Computer Science Department<br>Carnegie-Mellon University<br>Pittsburgh, PA 15213 | 1 |
| 9. | Dempsey Butler, III<br>1183 Arroyo Drive<br>Pebble Beach, CA 93953 | 1 |